Array Declarations

An object is declared to be an array with the `dimension` attribute.

```
logical, dimension (-99:99) :: yes_no
real, dimension (1:9, 0:9) :: x, y
integer, dimension (:,:,:) :: flags
```

Learn more about array declarations.

Next slide

Rank

The *rank* of an object is the number of dimensions. `yes_no` has rank 1. `x` and `y` have rank 2. `flags` has rank 3. A scalar has rank 0.

Previous slide Next slide

Shape

The *shape* of an object is a list of integers, as many as the rank, indicating the number of elements along each dimension. `yes_no` has shape (199). `x` and `y` have shape (9,10). `flags` has either deferred shape or assumed shape, depending on the context of the declaration. A scalar has shape (), which is an empty list. The `shape` intrinsic function returns the shape of an object.

Previous slide Next slide

Conformability

Two objects are *conformable* if they have the same shape or one is a scalar. Conformability is required in various situations, such as assignment.

Previous slide Next slide

Array Assignment

The right side is evaluated and stored into the variable on the left.

```
real, dimension (40) :: a, b, c
   . . .
a = 0
b = . . .; c = . . .
   . . .
a = a + 3.7 * b * abs (c)
```

Previous slide Next slide

Intrinsic Array Functions

Most intrinsic functions that were in Fortran 77 may have an array argument and the function is performed on each element of the array (elementally). Some new intrinsic functions are not elemental.

f = sum of a_i x cos x_i

```
f = sum (a * cos (x))
```

s = sum of the positive elements of vector a

```
s = sum (a, mask = (a > 0))
```

Learn more about intrinsic array functions.

Previous slide Next slide

where Statement and Construct

```
real, dimension (40, 40) :: a, b
   . . .
where (b /= 0)  a = a / b
```

Previous slide Next slide

`elsewhere` Statement

The `elsewhere` statement permits array assignments to be done where the logical expression is false.

```
real, dimension (m,n) :: b
integer, dimension (m,n) :: a
   . . .
where (abs(b) > huge (b)/100.0)
   a = 3
elsewhere (abs(b) > 10.0*epsilon (b))
   a = 2
elsewhere
   a = 1
end where
```

Learn more about the `where` construct.

Previous slide Next slide

`forall` Construct and Statement (F95)

The `forall` statement indicates that computations may be executed in parallel.

```
forall (i=1:n, j=1:m)
   a(i,j) = i+j
end forall
```

Previous slide Next slide

A `forall` statement can be used to assign the elements of the array `b` of rank one to the diagonal of array `a`. This cannot be done with array section notation.

```
forall (i=1:n)
   a(i,i) = b(i)
end forall
```

Previous slide Next slide

The following are permitted in a forall body:

- assignment statements
- pointer assignment statements
- `where` constructs
- `forall` constructs

Previous slide Next slide

Sometimes it is desirable to exclude some elements from taking part in a calculation. An optional mask expression may appear in a `forall` header. For example,

```
forall (i=1:n, j=1:m, &
        a(i)<9.0 .and. b(j)<9.0)
   c(i,j) = a(i) + b(j)
end forall
```

Learn more about the `forall` construct.

Previous slide Next slide

Triplet Notation

A part of an array may be designated with a triplet in place of a subscript. If `v` is rank one:

```
real, dimension (0:9) :: v
   . . .
v (0:3)    ! represents v(0), v(1), v(2), v(3)
v (3:7:2)  ! represents v(3), v(5), v(7)
v (7:)     ! represents v(7), v(8), v(9)
v (:1)     ! represents v(0), v(1)
v (:4:2)   ! represents v(0), v(2), v(4)
v (::5)    ! represents v(0), v(5)
v (6:1:-2) ! represents v(6), v(4), v(2)
```

Previous slide Next slide

a (3, 2:5) ! Shape is (4)

```
0  0  0  0  0
0  0  0  0  0
0  X  X  X  X
0  0  0  0  0
0  0  0  0  0
```

Previous slide Next slide

a (:, 2) ! Shape is (5)

```
0 X 0 0 0
0 X 0 0 0
0 X 0 0 0
0 X 0 0 0
0 X 0 0 0
```

Previous slide Next slide

a (2::2, 1:1) ! Shape is (2, 1)

```
0  0  0  0  0
X  0  0  0  0
0  0  0  0  0
X  0  0  0  0
0  0  0  0  0
```

Previous slide Next slide

Array Constructor

An array constructor builds an array from a collection of values. The values may be:

1. A scalar expression as in

   ```
   real, dimension (4) :: x
   x = (/ 1.2, 3.5, 1.1, 1.5 /)
   ```

   Previous slide Next slide

- An array expression as in

```
x = (/ a (i, 1:2), a (i+1, 2:3) /)
```

- An implied `do` loop as in

```
x = (/ (sqrt (real (i)), i = 1, 4) /)
```

All values of the components must have the same type and type parameters (kind and length). The rank of an array constructor is always one; however, the `reshape` intrinsic function can be used to define rank-two and greater arrays from the array constructor values.

Learn more about array constructors.

Previous slide Next slide

Reshape Intrinsic Function

Suppose we want to construct the integer array:

```
|  1  2  |
|  3  4  |
```

The array `a1234` can be declared and initialized to this value by either of the following:

```
integer, dimension(2,2) :: a1234 = reshape &
   ( (/ 1, 3, 2, 4 /), shape(a1234) )

integer, dimension(2,2) :: a1234 = reshape &
   ( (/ 1, 2, 3, 4 /), shape(a1234), &
     order = (/ 2, 1 /) )
```

Previous slide Next slide

Vector Subscripts

A *vector subscript* is a one-dimensional array of integers that can be used to select elements from an array.

```
real, dimension (0:9) :: v
integer, dimension (3) :: iv
   . . .
v = (/ (1.1*i, i = 0,9) /)
iv = (/ 3, 7, 2 /)
print *, v (iv)  ! Prints 3.3, 7.7, 2.2
```

Learn more about vector subscripts.

Previous slide Next slide

```
v = (/ (i, i = 1,5) /)
v (2:5) = v (1:4)
! result is v = ( 1, 1, 2, 3, 4 )
! not the same as a do loop
```

Previous slide Next slide

Exercise

1. If a chess or checkers board is declared by

   ```
   character (len=1), dimension (8, 8) :: board
   ```

   the statement

   ```
   board = "R"
   ```

   assigns the color red (``R") to all 64 positions.

   Previous slide Next slide

Write a statement or statements that assigns ``B" to the black positions. Assume that `board(1,1)` is to be red so that the board is as shown:

```
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
```

Previous slide Next slide

Allocatable Arrays

An array that is not a dummy argument may be declared with subscripts ``:" and the `allocatable` attribute. The rank of such an array is fixed, but the subscript bounds may be determined when the array is allocated during execution of the program.

Learn more about allocatable arrays.

Previous slide Next slide

```
real, dimension (:,:), allocatable :: amx
integer n
   . . .
read *, n
allocate (amx (n,n), stat = alloc_status)
   . . .
```

The value of `alloc_status` will be positive if the allocation was not successful.

Previous slide Next slide

Assumed-Shape Arrays

Dummy arguments declared with subscripts ``:" assume the shape of the actual argument passed. This ensures actual-dummy argument matching of shapes if the ranks match.

```
call ss (amx)
   . . .
subroutine ss (dummy_array)
   real, dimension (:,:) :: dummy_array
   . . .
```

Previous slide Next slide

Automatic Arrays

The size of an array in a procedure can depend on dummy arguments in various ways. For example, a local array is needed that is the same size as the dummy argument `a`.

```
subroutine s (a)
   real, intent (in), dimension (:) :: a
   real, dimension (size (a)) :: temp_a
   . . .
```

Learn more about automatic arrays.

Previous slide Next slide

More Intrinsic Functions

Three students take four exams. The results follow:

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55
```

Largest score:

```
maxval (score)  ! = 90
```

Previous slide Next slide

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55
```

Largest score for each student:

```
maxval (score, dim = 2)
   ! = (/ 90, 80, 66 /)
```

Previous slide Next slide

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55
```

Student with largest score:

```
maxloc (maxval (score, dim = 2))
   ! = maxloc ( (/ 90, 80, 66 /) )
   ! = (/ 1 /)
```

Previous slide Next slide

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55
```

Average score:

```
average = sum (score) / size (score)
      ! = 62


!!! Note that ! starts a comment !!!
```

Previous slide Next slide

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55
```

Number of scores above average:

```
above = score > average

   !    T T T F
   ! =  T F F T
   !    T F F F

count (above)  ! = 6
```

Previous slide Next slide

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55

above = score > average

   !    T T T F
   ! =  T F F T
   !    T F F F

count (above)  ! = 6
```

Did any student always score above the overall average?

```
any (all (above, dim = 2))  ! = .false.
```

E student A tests [score(student,test) > average]

Previous slide Next slide

```
          85  76  90  60
score =   71  45  50  80
          66  45  21  55

above = score > average

   !    T T T F
   ! =  T F F T
   !    T F F F

count (above)  ! = 6
```

Was there any test for which all students scored above the overall average?

```
any (all (above, dim = 1))  ! = .true.
```

E test A students [score(student,test) > average]

Learn more about intrinsic array functions.

Previous slide Next slide

Arrays of Random Numbers

What is the probability that a throw of two dice will yield a 7 or an 11? This program uses the built-in subroutine `random_number` to generate an array of random numbers between 0 and 1.

Previous slide Next slide

```
program seven_11

   implicit none
   integer, parameter :: number_of_rolls = 1000
   integer, dimension (number_of_rolls) ::  &
         dice, die_1, die_2
   integer :: wins

   call random_int (die_1, 1, 6)
   call random_int (die_2, 1, 6)
   dice = die_1 + die_2
   wins = count ((dice == 7) .or. (dice == 11))

   print "(a, f6.2)",  &
   "The percentage of rolls that are 7 or 11 is", &
   100.0 * real (wins) / real (number_of_rolls)
```

Previous slide Next slide

```
contains

subroutine random_int (result, low, high)

   integer, dimension (:), intent (out) :: result
   integer, intent (in) :: low, high
   real, dimension (size(result)) :: uniform_value

   call random_number (uniform_value)
   result =  &
   int ((high - low + 1) * uniform_value + low)

end subroutine random_int

end program seven_11
```

Learn more about intrinsic subroutines.

Previous slide Next slide

The built-in function `count` returns the number of true values in any logical array; in this case the value in the array is true if the corresponding value in the array `dice` is 7 or 11.

Previous slide Next slide

Exercise

1. Use the array version of `random_int` to write a simulation program to determine the percentage of times exactly five coins come up heads and five come up tails when ten fair coins are tossed simultaneously.

Previous slide Next slide

Date and Time

Two other useful intrinsic subroutines are `date_and_time` and `cpu_time` (F95).

`date_and_time` can return the date and time as a character string or using numerical values.

`cpu_time` returns a real value in seconds.

Previous slide Next slide

```
program time_it

  implicit none
  integer, parameter :: n = 1000000
  integer :: i, k1, k2, k3, k4
  character (len=8) :: date
  real :: start_time, stop_time
  real, dimension (n) :: a
  real, dimension (n) :: initial

  initial = (/ (1.1*i, i=1,n) /)

  call date_and_time (date = date)
  print *, "Date: ", date(1:4), "-", date(5:6), "-", date(7:8)
```

Previous slide Next slide

```
   k1 = 1; k2 = n-1
   k3 = 2; k4 = n

   a = initial

   call cpu_time (start_time)
   do i = k1, k2
     a(i) = a(i+1)
   end do
   call cpu_time (stop_time)

   print*
   print*, sum(a)
   print*, "CPU time for do loop is ", &
      stop_time - start_time, " seconds."

end program time_it
```

Previous slide Next slide

Another statement is timed, where the `do` loop is replaced by the assignment using triplet notation:

```
   a(k1:k2) = a(k3:k4)
```

Running the code on the Fujitsu compiler and a 333Mhz Sun Ultra produced the following times (in seconds):

```
              Loop      Triplet

Unoptimized   0.95         2.70

Optimized     0.34         0.25
```

Previous slide